Random Slicing: Efficient and Scalable Data Placement for Large-scale Storage Systems

ALBERTO MIRANDA, Barcelona Supercomputing Center, Barcelona
SASCHA EFFERT, Christmann Informationstechnik + Medien, Ilsede
YANGWOOK KANG and ETHAN L. MILLER, University of California, Santa Cruz
IVAN POPOV and ANDRE BRINKMANN, Johannes Gutenberg-University, Mainz
TOM FRIEDETZKY, Durham University, Durham
TONI CORTES, Barcelona Supercomputing Center and Technical University of Catalonia, Barcelona

The ever-growing amount of data requires highly scalable storage solutions. The most flexible approach is to use storage pools that can be expanded and scaled down by adding or removing storage devices. To make this approach usable, it is necessary to provide a solution to locate data items in such a dynamic environment. This paper presents and evaluates the Random Slicing strategy, which incorporates lessons learned from table-based, rule-based, and pseudo-randomized hashing strategies and is able to provide a simple and efficient strategy that scales up to handle exascale data. Random Slicing keeps a small table with information about previous storage system insert and remove operations, drastically reducing the required amount of randomness while delivering a perfect load distribution.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*

General Terms: Design, Experimentation, Performance, Reliability, Scalability

Additional Key Words and Phrases: PRNG, Randomized data distribution, Storage management

ACM Reference Format:
Miranda, A., Effert, S., Kang Y., Miller, E. L., Popov, I., Brinkmann, A., Friedetzky, T. and Cortes, T. 2012. Random Slicing: Efficient and Scalable Data Placement for Large-scale Storage Systems. ACM Trans. Storage V, N, Article A (January YYYY), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

An earlier version of this article appeared in *Proceedings of the 18th International Conference on High Performance Computing (HiPC)* [Miranda et al. 2011].
This work was partially supported by the Spanish Ministry of Science and Technology under the TIN2007-60625 grant, the Catalan Government under the 2009-SGR-980 grant, the EU Marie Curie Initial Training Network SCALUS under grant agreement no. 238808, the National Science Foundation under grants CCF-0937938 and IIP-0934401, and by the industrial sponsors of the Storage Systems Research Center at the University of California, Santa Cruz.
Authors' addresses: A. Miranda and T. Cortes, Barcelona Supercomputing Center, 31 Jordi Girona, Barcelona, 08034, Spain; email: alberto.miranda@bsc.es, toni.cortes@bsc.es; S. Effert, Christmann Informationstechnik + Medien, 10 Ilseder Htte, 31241, Ilsede, Germany; email: sascha.effert@christmann.info; Y. Kang and E. L. Miller, University of California, Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA; email: ywkang@ucsc.edu, elm@ucsc.edu; I. Popov and A. Brinkmann, Johannes Gutenberg-Universitt Mainz, Zentrum für Datenverarbeitung (ZDV), Anselm-Franz-von-Bentzel-Weg 12, D 55099 Mainz, Germany; email: ipopov@uni-mainz.de, brinkman@uni-mainz.de; T. Friedetzky, Durham University, Science Labs, South Road, Durham, DH1 3LE, England, U.K.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The ever-growing creation of, and demand for, massive amounts of data requires highly scalable storage solutions. The most flexible approach is to use a pool of storage devices that can be expanded and scaled down as needed by adding new storage devices or removing older ones; this approach necessitates a scalable solution for locating data items in such a dynamic environment.

Table-based strategies can provide an optimal mapping between data blocks and storage systems, but obviously do not scale to large systems because tables grow linearly in the number of data blocks. Rule-based methods, on the other hand, run into fragmentation problems, so defragmentation must be performed periodically to preserve scalability.

Hashing-based strategies use a compact function h in order to map balls with unique identifiers out of some large universe U into a set of bins called S so that the balls are evenly distributed among the bins. In our case, balls are data items and bins are storage devices. Given a static set of devices, it is possible to construct a hash function so that every device gets a fair share of the data load. However, standard hashing techniques do not adapt well to a changing set of devices.

Consider, for example, the hash function $h(x) = (a \cdot x + b) \bmod n$, where S represents the set of storage devices and is defined as $S = \{0, \ldots, n-1\}$. If a new device is added, we are left with two choices: either replace n by $n+1$, which would require virtually all the data to be relocated; or add additional rules to $h(x)$ to force a certain set of data blocks to be relocated on the new device in order to get back to a fair distribution, which, in the long run, destroys the compactness of the hashing scheme.

Pseudo-randomized hashing schemes that can adapt to a changing set of devices have been proposed and theoretically analyzed. The most popular is probably Consistent Hashing [Karger et al. 1997], which is able to evenly distribute single copies of each data block among a set of storage devices and to adapt to a changing number of disks. We will show that these pure randomized data distribution strategies have, despite their theoretical perfectness, serious drawbacks when used in very large systems. Especially, many of their properties are only achieved if enough random experiments can be performed. This has, as we will show in this paper, a serious influence on the necessary amount of memory, rendering them infeasible in large scale environments.

Besides adaptivity and fairness, redundancy is important as well. Storing just a single copy of a data item in real systems is dangerous because, if a storage device fails, all of the blocks stored in it are lost. It has been shown that simple extensions of standard randomized data distribution strategies to store more than a single data copy are not always capacity efficient [Brinkmann et al. 2007].

The main contributions of this paper are:

— **First comparison of different hashing-based data distribution strategies that are able to replicate data in a heterogeneous and dynamic environment**. This comparison shows the strengths and drawbacks of the different strategies as well as their constraints. Such comparison is novel because hashing-based data distribution strategies have been mostly analytically discussed, with only a few implementations available, and in the context of peer-to-peer networks with limited concern for the fairness of the data distribution [Stoica et al. 2003]. Only a few of these strategies have been implemented in storage systems, where limited fairness immediately leads to a strong increase in costs [Brinkmann et al. 2004][Weil et al. 2006a].

— The introduction of **Random Slicing, which overcomes the drawbacks of randomized data distribution strategies** by incorporating lessons learned from table-based, rule-based and pseudo-randomized hashing strategies. Random Slicing keeps a small table with information about previous storage system insertions and removals. This table helps to drastically reduce the required amount of randomness in the system and thus reduces the amount of necessary main memory by several orders of magnitude.

— An analysis of the **influence** (or lack thereof) **of eighteen different pseudo-random number generators (PRNGs) in all the strategies studied**. We evaluate each generator in terms of quality of distribution and performance provided. We believe that the results of this analysis may help other researchers choose an appropriate PRNG when designing and implementing new randomized data distribution strategies.

It is important to note that all randomized strategies map (virtual) addresses to a set of disks, but do not define the placement of the corresponding block on the disk surface. This placement on the block devices has to be resolved by additional software running on the disk itself. Therefore, we will assume inside the remainder of the paper that the presented strategies work in an environment that uses object-based storage. Unlike conventional block-based hard drives, object-based storage devices (OSDs) manage disk block allocation internally, exposing an interface that allows others to read and write to variably-sized, arbitrarily-named objects [Azagury et al. 2003][Devulapalli et al. 2008].

1.1. The Model

Our research is based on an extension of the standard "balls into bins" model [Johnson and Kotz 1977][Mitzenmacher 1996]. Let $\{0, \ldots, M-1\}$ be the set of all identifiers for the balls and $\{0, \ldots, N-1\}$ be the set of all identifiers for the bins. Suppose that the current number of balls in the system is $m \leq M$ and that the current number of bins in the system is $n \leq N$. We will often assume for simplicity that the balls and bins are numbered in a consecutive way starting with 0, but any numbering that gives unique numbers to each ball and bin would work for our strategies.

Suppose that bin i can store up to b_i (copies of) balls. Then we define its relative capacity as $c_i = b_i / \sum_{j=0}^{n-1} b_j$. We require that, for every ball, k copies must be stored in different bins for some fixed k. In this case, a trivial upper bound for the number of balls the system can store while preserving fairness and redundancy is $\sum_{j=0}^{n-1} b_j / k$, but it can be much less than that in certain cases. We term the k copies of a ball a *redundancy group*.

Placement schemes for storing redundant information can be compared based on the following criteria (see also [Brinkmann et al. 2002]):

— *Capacity Efficiency and Fairness.* A scheme is called *capacity efficient* if it allows us to store a near-maximum number of data blocks. We will see in the following that the fairness property is closely related to capacity efficiency, where *fairness* describes the property that the number of balls *and* requests received by a bin are proportional to its capacity.

— *Time Efficiency.* A scheme is called *time efficient* if it allows a fast computation of the position of any copy of a data block without the need to refer to centralized tables. Schemes often use smaller tables that are distributed to each node that must locate blocks.

— *Compactness.* We call a scheme *compact* if the amount of information the scheme requires to compute the position of any copy of a data block is small (in particular, it should only depend on n—the number of bins).

— *Adaptivity.* We call a scheme *adaptive* if it only redistributes a near-minimum amount of copies when new storage is added in order to get back into a state of fairness.

We therefore compare the different strategies in Section 6 with the minimum amount of movements, which is required to keep the fairness property.

Our goal is to find strategies that perform well under all of these criteria.

1.2. Previous Results

Data reliability and support for scalability as well as the dynamic addition and removal of storage systems is one of the most important issues in designing storage environments. Nevertheless, up to now only a limited number of strategies has been published for which it has formally been shown that they can perform well under these requirements.

Data reliability is achieved by using RAID encoding schemes, which divide data blocks into specially encoded sub-blocks that are placed on different disks to make sure that a certain number of disk failures can be tolerated without losing any information [Patterson et al. 1988]. RAID encoding schemes are normally implemented by striping data blocks according to a pre-calculated pattern across all the available storage devices. Even though deterministic extensions for the support of heterogeneous disks have been developed [Cortes and Labarta 2001][Gonzalez and Cortes 2008], adapting the placement to a changing number of disks is cumbersome under RAID as all of the data may have to be reorganized.

In the following, we just focus on data placement strategies that are able to cope with dynamic changes of the capacities or the set of storage devices in the system. Karger, *et al.* present an adaptive hashing strategy for homogeneous settings that satisfies fairness and is 1-competitive w.r.t. adaptivity [Karger et al. 1997]. In addition, the computation of the position of a ball takes only an expected number of $O(1)$ steps. However, their data structures need at least $n \log^2 n$ bits to ensure a good data distribution.

Brinkmann, *et al.* presented the cut-and-paste strategy as alternative placement strategy for uniform capacities [Brinkmann et al. 2000]. Their scheme requires $O(n \log n)$ bits and $O(\log n)$ steps to evaluate the position of a ball. Furthermore, it keeps the deviation from a fair distribution of the balls extremely small with high probability. Interestingly, the theoretical analysis of this strategy has been experimentally re-evaluated in a recent paper by Zheng *et al.* [Zheng and Zhang 2011].

Sanders considers the case that bins fail and suggests to use a set of forwarding hash functions $h_1, h_2, \ldots, h_k$, where at the time h_i is set up, only bins that are intact at that time are included in its range [Sanders 2001].

Adaptive data placement schemes that are able to cope with arbitrary heterogeneous capacities have been introduced in [Brinkmann et al. 2002]. The presented strategies Share and Sieve are compact, fair, and (amortized) $(1 + \epsilon)$-competitive for arbitrary changes from one capacity distribution to another, where $\epsilon > 0$ can be made arbitrarily small. Other data placement schemes for heterogeneous capacities are based on geometrical constructions [Schindelhauer and Schomaker 2005]; the linear method used combines the standard consistent hashing approach [Karger et al. 1997] with a linear weighted distance measure. By using bin copies and different partitions of the hash space, the scheme can get close to a fair distribution with high probability. A second method, called *logarithmic method*, uses a logarithmic distance measure between the bins and the data to find the corresponding bin.

All previously mentioned work is only applicable for environments where no replication is required. Certainly, it is easy to come up with proper extensions of the schemes so that no two copies of a ball are placed in the same bin. A simple approach feasible for all randomized strategies to replicate a ball k times is to perform the experiment k times and to remove after each experiment the selected bin. Nevertheless, it has been shown that the fairness condition cannot be guaranteed for these simple strategies and that capacity will be wasted [Brinkmann et al. 2007]. This paper will also evaluate the influence of this capacity wasting in realistic settings.

The first methods with dedicated support for replication were proposed by Honicky and Miller [Honicky and Miller 2003][Honicky and Miller 2004]. *RUSH* (Replication Under Scalable Hashing) maps replicated objects to a scalable collection of storage servers according to user-specified server weighting. When the number of servers changes, RUSH tries to redistribute as few objects as possible to restore a balanced data distribution while ensuring that no two replicas of an object are ever placed on the same server.

A drawback of the RUSH-variants is that they require that new capacity is added in chunks, where each chunk is based on servers of the same type and the number of disks inside a chunk has to be sufficient to store a complete redundancy group without violating fairness and redundancy. The use of sub-clusters is required to overcome the problem if more than a single block of a redundancy group is accidentally mapped to the same hash-value. This property leads to restrictions for bigger numbers of sub-blocks. In this case, prime numbers can be used to guarantee a unique mapping between blocks of a redundancy group and the servers inside a chunk.

CRUSH is derived from RUSH and supports different hierarchy levels that provide the administrator finer control over the data placement in the storage environment [Weil et al. 2006b]. The algorithm accommodates a wide variety of data replication and reliability mechanisms and distributes data in terms of user-defined policies.

Amazon's Dynamo [DeCandia et al. 2007] uses a variant of Consistent Hashing with support for replication where each node is assigned multiple tokens chosen at random that are used to partition the hash space. This variant has given good results concerning performance and fairness, though the authors claim that it might have problems scaling up to thousands of nodes.

Brinkmann *et al.* have shown that a huge class of placement strategies cannot preserve fairness and redundancy at the same time and have presented a placement strategy for an arbitrary fixed number k of copies for each data block, which is able to run in $O(k)$. The strategies have a competitiveness of $\log n$ for the number of replacements in case of a change of the infrastructure [Brinkmann et al. 2007]. This competitiveness has been reduced to $O(1)$ by breaking the heterogeneity of the storage systems [Brinkmann and Effert 2008]. Besides the strategies presented inside this paper, it is worth mentioning the Spread strategy, which has similar properties to those of Redundant Share [Mense and Scheideler 2008].

To the best of our knowledge there is only one structured analysis of the influence of pseudo-random number generators in data distribution strategies. Popov *et al.* replace the internal hash function of Consistent Hashing and Redundant Share with several pseudo-random number generators and evaluate the strategies both in terms of performance and quality of distribution [Popov et al. 2012]. We decided to extend this evaluation to all the strategies that we considered in the paper.

2. RANDOMIZED DATA DISTRIBUTION

We present in this section a short description of the applied data distribution strategies. We start with Consistent Hashing and Share, which can, in their original form, only be applied for $k = 1$ (i.e. only one copy of each data block) and therefore lack support for redundancy. Both strategies are used as sub-strategies inside some of the investigated data distribution strategies. Besides their usage as sub-strategies, we will also present a simple replication strategy, which can be based on any of these simple strategies. Afterwards, we present Redundant Share and RUSH, which directly support data replication.

2.1. Consistent Hashing

We start with the description of the Consistent Hashing strategy, which solves the problem of (re-)distributing data items in homogeneous systems [Karger et al. 1997]. In Consistent Hashing, both data blocks and storage devices are hashed to random points in a $[0, 1)$-interval, and the storage device closest to a data block in this space is responsible for that data block. Consistent Hashing ensures that adding or removing a storage device only requires a near minimal amount of data replacements to get back to an even distribution of the load. However, the consistent hashing technique cannot be applied well if the storage devices can have arbitrary non-uniform capacities since in this case the load distribution has to be adapted to the capacity distribution of the devices. The memory consumption of Consistent Hashing heavily depends on the required fairness. Using only a single point for each storage devices leads to a load deviation of $n \cdot \log n$ between the least and heaviest loaded storage devices. Instead it is necessary to use $\log n$ virtual devices to simulate each physical device, respectively to throw $\log n$ points for each device to achieve a constant load deviation.

2.2. Share-strategy

Share supports heterogeneous environments by introducing a two stage process [Brinkmann et al. 2002]. In the first stage, the strategy randomly maps one interval for each storage system to the $[0, 1)$-interval. The length of these intervals is proportional to the size of the corresponding storage systems and some stretch factor s and can cover the $[0, 1)$-interval many times. In this case, the interval is represented by several virtual intervals. The data items are also randomly mapped to a point in the $[0, 1)$-interval. Share now uses an adaptive strategy for homogeneous storage systems, like Consistent Hashing, to get the responsible storage systems from all storage systems for which the corresponding interval includes this point.

The analysis of the Share-strategy shows that it is sufficient to have a stretch factor $s = O(\log N)$ to ensure correct functioning and that Share can be implemented in expected time $O(1)$ using a space of $O(s \cdot k \cdot (n + 1/\delta))$ words (without considering the hash functions), where δ characterizes the required fairness. Share has an amortized competitive ratio of at most $1 + \epsilon$ for any $\epsilon > 0$. Nevertheless, we will show that, similar to Consistent Hashing, the memory consumption heavily depends on the expected fairness.

2.3. Trivial data replication

Consistent Hashing and Share are, in their original setting, unable to support data replication or erasure codes, since it is always possible that multiple strips belonging to the same stripe set are mapped to the same storage system and that data recovery in case of failures becomes impossible. Nevertheless, it is easy to imagine strategies to overcome this drawback and to support replication strategies by, e.g., simply removing all previously selected storage systems for the next random experiment for a stripe set. Another approach, used inside the experiments in this paper, is to simply perform as many experiments as are necessary to get enough independent storage systems for the stripe set. It has been shown that this trivial approach wastes some capacity [Brinkmann et al. 2007], but we will show in this paper that this amount can often be neglected.

2.4. Redundant Share

Redundant Share has been developed to support the replication of data in heterogeneous environments. The strategy orders the bins according to their weights c_i and sequentially iterates over the bins [Brinkmann et al. 2007]. The basic idea is that the weights are calculated in a way that ensures perfect fairness for the first copy and to use a recursive descent to select additional copies. Therefore, the strategy needs $O(n)$ rounds for each selection process. The algorithm is $\log n$-competitive concerning the number of replacements if storage systems enter or leave the system. The authors of the original strategy have also presented extensions of Redundant Share, which are $O(1)$-competitive concerning the number of replacements [Brinkmann and Effert 2008] as well as strategies, which have $O(k)$-runtime. Both strategies rely on Share and we will discuss in the evaluation section why they are not feasible in realistic settings.

2.5. RUSH

The RUSH algorithms all proceed in two stages, first identifying the appropriate cluster in which to place an object, and then identifying the disk within a cluster [Honicky and Miller 2003][Honicky and Miller 2004]. Within a cluster, replicas assigned to the cluster are mapped to disks using prime number arithmetic that guarantees that no two replicas of a single object can be mapped to the same disk. The selection of clusters is a bit more complex and differs between the three RUSH variants: $RUSH_P$, $RUSH_R$, and $RUSH_T$. $RUSH_P$ considers clusters in the reverse of the order they were added, and determines whether an object would have been moved to the cluster when it was added; if so, the search terminates and the object is placed. $RUSH_R$ works in a similar way, but it determines the number of objects in each cluster simultaneously, rather than requiring a draw for each object. $RUSH_T$ improves the scalability of the system by descending a tree to assign objects to clusters; this reduces computation time to $\log c$, where c is the number of clusters added.

3. RANDOM SLICING

In this section we describe our proposal for a new data distribution strategy called Random Slicing. This strategy tries to overcome the drawbacks of current randomized data distribution strategies by incorporating lessons learned from table-based and pseudo-randomized hashing strategies. In particular, it tries to reduce the required amount of randomness necessary to keep a uniform distribution, which can cause memory consumption problems.

3.1. Description

Random Slicing is designed to be fair and efficient both in homogeneous and heterogeneous environments and to adapt gracefully to changes in the number of bins. Suppose that we have a random function $h : \{1, \ldots, M\} \rightarrow [0, 1)$ that maps balls uniformly at random to real numbers in the interval $[0, 1)$. Also, suppose that the relative capacities for the n given bins are $(c_0, \ldots, c_{n-1}) \in [0, 1)^n$ and that $\sum_{i=0}^{n-1} c_i = 1$.

The strategy works by dividing the $[0, 1)$ range into intervals and assigning them to the bins currently in the system (see Figure 1). Notice that the intervals created do not overlap and completely cover the $[0, 1)$ range. Also note that bin i can be responsible for several non-contiguous intervals $P_i = (I_0, \ldots, I_k)$, where $k < n$, which will form the *partition* of that bin. To ensure fairness, Random Slicing will always enforce that $\sum_{j=0}^{k-1} |I_j| = c_i$.

In an initial phase, i.e. when the first set of bins enters the system, each bin i is given only one interval of length c_i, since this suffices to maintain fairness. Whenever new bins enter the system, however, relative capacities for old bins change due to the increased overall capacity. To maintain fairness, Random Slicing shrinks existing partitions by splitting the intervals that compose them until their new relative capacities are reached. The new intervals generated are used to create partitions for the new bins.

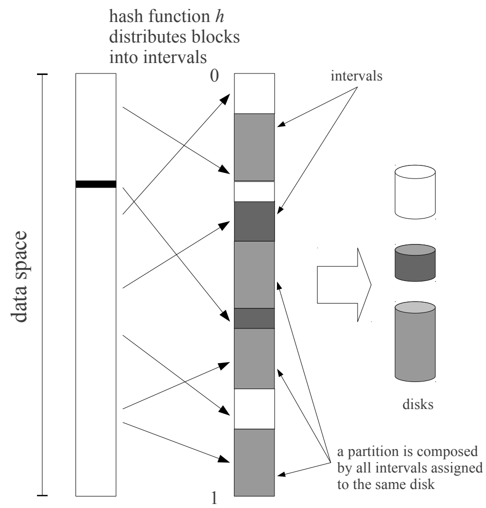


Fig. 1. Overview of Random Slicing's data distribution. The data space is divided into intervals that are assigned to storage devices. A storage device can be responsible of several intervals that configure the device's partition of the data space. The size of each partition is computed so that the amount of data contained in it matches the capacity of the device relative to the overall storage capacity. Blocks are assigned to intervals using a pseudo-randomized hash function that guarantees uniformity of distribution.

First, the strategy computes by how much partitions should be reduced in order to keep the fairness of the distribution. Since the global capacity has increased, each partition P_i must be reduced by $r_i = c_i - c'_i$, where c'_i corresponds to the new relative capacity of bin i.

Partitions become smaller by releasing or splitting some of their intervals, thus generating *gaps*, which can be used for new intervals. This way, the partition lengths for the old bins already represent their corresponding relative capacities and it is only necessary to use these gaps to create new partitions for the newly added bins.

The time efficiency and the memory consumption of the strategy, of course, crucially depend on the maximum number of intervals being managed. In the following theorems we show that this number is low.

THEOREM 3.1. *Assume an environment where storage systems can only be added. In this case, adding n storage systems leads to at most $1/2n \cdot (n+1)$ intervals.*

PROOF. We use an induction technique to proof the theorem. For the base case it holds that adding the first storage system leads to 1 interval and it holds that $1/2 \cdot 1 \cdot 2 = 1$. In the following we show the inductive step from change n to change $(n+1)$. We show that adding a new storage system leads to at most n new intervals and that therefore the number of intervals after the insertion is at most $1/2n \cdot (n+1) + n = 1/2(n^2+3n) \leq 1/2(n^2+3n+2) = 1/2(n+1)(n+2)$, which is the inductive step.

The number of intervals is smaller than n before a new storage system is added. Then, a part of the intervals of each storage system already within the system has to be assigned to the new storage system. Assume now for each existing storage system that we pick an arbitrary interval first. If the size of this interval is smaller than the interval length, which has to be assigned to the new storage system, we assign the complete interval to the new storage system. The number of intervals does not change in this case and we continue with the next intervals until we reach one interval which cannot be completely assigned to the new storage system. In this case, we split this interval and assign one part to the new storage system and keep the other for the previously existing one. Therefore, the number of interval increases by at most one for each previously existing storage system (or stays constant if the last assigned interval will be completely assigned to the new storage system)[1]. □

In the following we investigate an environment where storage systems can also be removed. We assume in the following that the number of storage systems is not decreasing for a long time in a typical storage environment and that new storage systems are always bigger than storage systems which have been removed. We call a situation, where at most as many storage systems, as have been previously removed, have been added to the environment steady state.

THEOREM 3.2. *Assume that there have been at most n storage systems in the environment. Then the number of intervals will be small $1/2n \cdot (n+1)$ in the steady state.*

PROOF. Assume that k storage system have been removed and no new storage systems have been added. In this case it can occur that the number of intervals becomes bigger than $1/2n \cdot (n+1)$. Now assume that at k new storage systems have been added to the environment and the system is back in a steady state. We will show in this case that the number of intervals is at most $1/2n \cdot (n+1)$, where n is the number of all storage systems, which have been previously been part of the environment. We use a technique inspired by the zero-height-strategy proposed in [Brinkmann et al. 2000] and assign first all intervals, which have been previously assigned to the removed storage system, to the new storage systems. This is possible as the new storage systems are at least as big as the removed storage systems. If the new storage systems are bigger than the removed ones, additional intervals lengths have to be assigned to them. In this case, we just assign the remaining capacity in the same way as performed in Theorem 3.1. □

3.2. Interval Creation Algorithm

The goal of the interval creation algorithm is to split existing intervals in a way that allows new intervals to be created while maintaining the relative capacities of the system. Note that the strategy's memory consumption directly depends on the number of intervals used and, therefore, the number of new intervals created in each addition phase can hamper scalability. We briefly explain two interval creation strategies and two variants.

— **Greedy:** This algorithm tries to collect as many complete intervals as possible and will only split an existing interval as a last resort. Furthermore, when splitting an interval is the only option, the algorithm tries to expand any adjacent gap instead of creating a new one. Once enough gaps are collected to produce an even distribution, they are assigned sequentially to new partitions.

— **CutShift:** This algorithm also tries to collect as many complete intervals as possible, but, when it is necessary to split an interval, alternates between splitting it by the beginning or by the end in an attempt to maximize gap length. Once enough gaps are collected to produce an even distribution, they are assigned sequentially to new partitions.

— **Greedy+Sorted:** A variant of Greedy where the largest partitions are assigned greedily to the largest gaps available.

— **CutShift+Sorted:** A variant of CutShift where the largest partitions are assigned greedily to the largest gaps available.

[1]Notice, however, that the theorem assumes a worst-case scenario where storage systems are added one by one and that, as we will see in Section 5, an appropriate algorithm can further reduce the number of intervals if several storage systems are added in bulk.

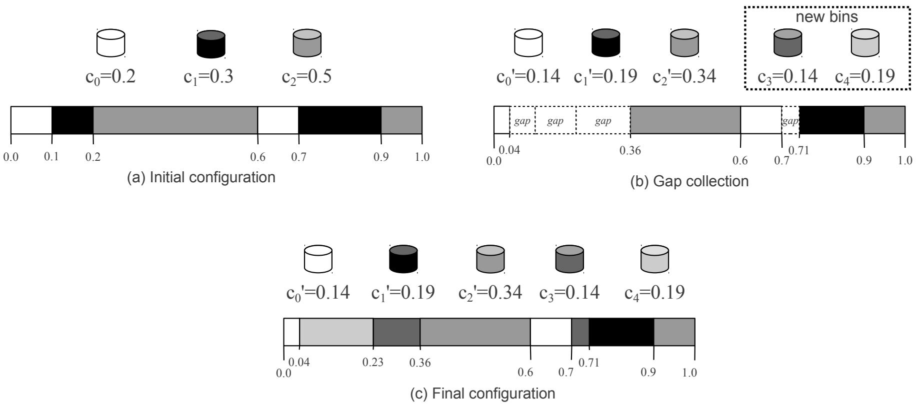


Fig. 2. Example of the CutShift+Sorted algorithm. Two new devices B_3 and B_4 are added to the system, representing a capacity increase of 14% and 19%, respectively. The algorithm computes the new relative capacities (c_i) of each partition and proceeds to create *gaps* by either assimilating or slicing intervals. Interval $[0.1, 0.2)$ is completely assimilated, while intervals $[0.0, 0.1)$ and $[0.2, 0.6)$ are cut. Note how the algorithm shifts the cutting point to the beginning or the end of the interval to increase the length of the gap. When all gaps are created the largest partitions are assigned to the largest intervals.

Note that these strategies are intentionally simple since our intention is that interval reorganizations can be computed as fast as possible in order to reduce the reconfiguration time of the storage system.

An example of the CutShift+Sorted reorganization is shown in Figure 2, where two new bins B_3 and B_4, representing a 33% capacity increase, are added to the bins B_0, B_1, and B_2. Figure 2(a) shows the initial configuration and the relative capacities for the initial bins. Figure 2(b) shows that the partition of B_0 must be reduced by 0.06, the partition of B_1 by 0.11, and the one of B_2 by 0.16, whereas two new partitions with a size of 0.14 and 0.19 must be created for B_3 and B_4. The interval $[0.1, 0.2) \in B_1$ can be completely cannibalized, whereas the intervals $[0.0, 0.1) \in B_0$, $[0.2, 0.6) \in B_2$ and $[0.7, 0.9) \in B_1$ are split while trying to maximize gap lengths. Figure 2(c) shows that the partition for B_3 is composed of intervals $[0.23, 0.36)$ and $[0.7, 0.71)$, while the partition for B_4 consists only of interval $[0.04, 0.23)$.

3.3. Data Lookup

Once all partitions are created, the location of a data item/ball b can be easily determined by calculating $x = h(b)$ and retrieving the bin associated with it. Notice that some balls will change partition after the reorganization, but as partitions always match their ideal capacity, only a near minimal amount of balls will need to be reallocated. Furthermore, if $h(b)$ is uniform enough and the number of balls in the system significantly larger than the number of intervals (both conditions easily feasible), the fairness of the strategy is guaranteed.

4. METHODOLOGY

Most previous evaluations of data distribution strategies are based on an analytical investigation of their properties. In contrast, we will use a simulation environment to examine the real-world properties of the investigated protocols. The simulation environment has been developed by the authors of this paper and is available online[2].

[2]http://dadisi.sourceforge.net

This collection also includes the parameter settings for the individual experiments described in the paper.

First, we evaluate the scalability of Random Slicing with the different interval creation algorithms proposed in Section 3.2. This way, we will determine how well does the strategy scale and we will select the best algorithm for the rest of the simulations.

Second, we run experiments for all distribution strategies described in Section 2 in an environment that scales from a few storage systems up to thousands of devices. We evaluate these strategies in terms of fairness, adaptivity, performance and memory consumption and compare the results with those of Random Slicing.

Third, we measure the impact of the randomization function in distribution quality and performance. We run several simulations where we change the pseudo-random number generator used by each strategy. Each experiment measures the changes in fairness and performance of each strategy.

All experiments assume that each storage node (also called storage system in the following) consists of 16 hard disks (plus potential disks to add additional intra-shelf redundancy). Besides, most experiments distinguish between an homogeneous setting (all devices have the same capacity) and an heterogeneous settings (the capacity of devices varies).

In all experiments described but those of Section 7, the implementations of Consistent Hashing, Share and Redundant Share use a custom implementation of the SHA1 pseudo-random number generator [Eastlake and Jones 2001]. The $RUSH^*$ variants use the William-Hill generator, while the implementation of Random Slicing uses Thomas Wang's 64 bit mixing function [Wang 2007].

5. SCALABILITY OF RANDOM SLICING

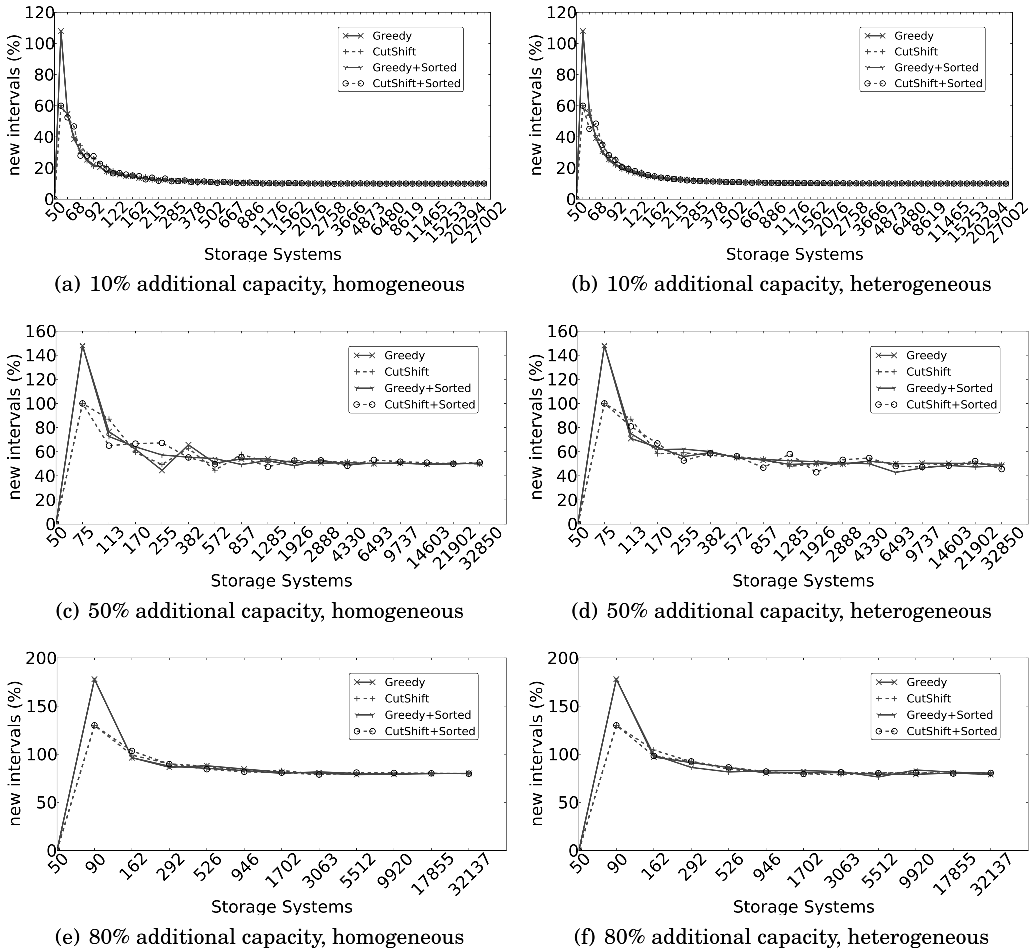
Random Slicing's performance and memory consumption largely depends on how well the number of intervals scales when there are changes in the storage system. An excessive amount of new intervals can render the strategy useless due to memory or performance constraints. In this section we evaluate the different interval creation algorithms proposed in Section 3.2 and measure how well they scale.

Figure 3 plots the percentage of new intervals created when adding devices to the storage systems. The evaluation begins with 50 devices and adds, in each step, as much new devices as needed to increase capacity by 10%, 50% or 80%. We evaluate a homogeneous setting where all the devices added have the same capacity, as well as a heterogeneous setting where the capacity of new devices increases by a factor of 1.5 in each step. In order to test long-term scalability, we continuously add devices until we surpass 25,000 devices.

Figures 3(a), 3(c) and 3(e) depict the evaluation results for the homogeneous setting, while Figures 3(b), 3(d) and 3(f) show the results for the heterogeneous setting. All four algorithms behave similarly in all experiments, which is to be expected as the number of intervals is not directly related with the capacity of the devices. All show an initial phase where adding devices leads to a high percentage of new intervals, with the Greedy algorithm even doubling the number of intervals in the first addition phase. This is not surprising since initially the number of intervals to work with is small, which limits the capability of the algorithms to create large gaps and increases the number of new intervals. This also explains why CutShift is more successful at reducing the number of new intervals than Greedy, since it's able to create larger gaps.

Assigning the largest partitions to the largest gaps also has some influence in reducing the number of new intervals, as the results for Greedy+Sorted and CutShift+Sorted show when comparing them against the non-sorted variants.

Note that once the algorithms have enough intervals to work with, they are significantly more effective: depending on the target increase in capacity, all strategies are



(a) 10% additional capacity, homogeneous

(b) 10% additional capacity, heterogeneous

(c) 50% additional capacity, homogeneous

(d) 50% additional capacity, heterogeneous

(e) 80% additional capacity, homogeneous

(f) 80% additional capacity, heterogeneous

Fig. 3. Percentage of new intervals created. Each step adds as many homogeneous or heterogeneous devices as necessary to reach the target capacity increase (i.e. 10%, 50%, and 80%). In the case of heterogeneous devices, the capacity of each device is increased by 1.5.

able to reduce the number of new intervals to around 10%, 50% or 80% per step, respectively, and keep it steady even after more than 10 reorganizations. Interestingly enough, the number of new intervals created roughly corresponds to the increase in capacity, regardless of the number of devices added.

Based on the above evaluation, we select the CutShift+Sorted as our management algorithm, which gives good results both in the initial and the stable phase. From now on, every evaluation of Random Slicing will use this algorithm. For the sake of completion, Algorithm 1 shows the pseudo-code for this mechanism.

6. EVALUATION OF RANDOMIZED STRATEGIES

The following section evaluates the impact of the different distribution strategies on the data distribution quality, the memory consumption of the different strategies, their adaptivity and performance. All graphs presented in the section contain four bars for each number of storage systems, which represent the experimental results for one, two, four, and eight copies (please see Figure 4 for the color codes in the legend). The

Algorithm 1 CutShift Gap Collection Algorithm

Input: $\{b_0, \ldots, b_{n-1}\}$, $\{b_n, \ldots, b_{p-1}\}$, $\{I_0, \ldots, I_{q-1}\}$
Output: $gaps : \{G_0, \ldots, G_{m-1}\}$
Require: $(p > n) \wedge (q \geq n)$

```
∀i ∈ {0, ..., p−1} : c'_i ← b_i / Σ_{j=1}^{p−1} b_j
∀i ∈ {0, ..., n−1} : r_i ← c_i − c'_i
gaps ← {}
for i ← 0 to q − 1 do
  j ← get bin assigned to I_i
  G ← get last gap from gaps
  if r_j > 0 then
    if length(I_i) < r_j then
      if adjacent(G, I_i) then
        G ← G + length(I_i)
      else
        gaps ← gaps + I_i
      end if
      r_i ← r_i − length(I_i)
      if last interval was assimilated completely then
        cut_interval_end ← false
      end if
    else
      if adjacent(G, I_i) then
        G ← G + length(I_i)
      else
        if cut_interval_end then
          gaps ← gaps + {I_i.end − r_j, I_i.end}
        else
          gaps ← gaps + {I_i.start, I_i.start + r_j}
        end if
      end if
      r_i ← r_i − r_j
      cut_interval_end ← ¬cut_interval_end
    end if
  end if
end for
return gaps
```

white boxes in each bar represent the range of results, e.g., between the minimum and the maximum usage. Also, the white boxes include the standard deviation for the experiments. Small or non-existing white boxes indicate a very small deviation between the different experiments.

We assume that each storage systems in the homogeneous setting can hold up to $k \cdot 500,000$ data items, where k is the number of copies of each block. Assuming a hard disk capacity of 1 TByte and putting 16 hard disks in each shelf means that each data item has a size of 2 MByte. The number of placed data items is $k \cdot 250,000$ times the number of storage systems. In all cases, we compare the fairness, the memory consumption, as well as the performance of the different strategies for a different number of storage systems.

In the heterogeneous setting we begin with 128 storage systems and we add 128 new systems in each step, each with $3/2$ times the size of the previously added system. We are placing again half the number of items, which saturates all disks.

For each of the homogeneous and heterogeneous tests, we also count the number of data items, which have to be moved in case we are adding disks, so that the data distribution delivers the correct location for a data item after the redistribution phase. The number of moved items has to be as small as possible to support dynamic environments, as the systems typically tend to a slower performance during the reconfiguration process.

We will show that the dynamic behavior can be different if the order of the k copies is important, e.g. in case of parity RAID, Reed-Solomon codes, or EvenOdd-Codes, or if this order can be neglected in case of pure replication strategies [Patterson et al. 1988][Blaum et al. 1994][Corbett et al. 2004].

6.1. Fairness

The first simulations evaluate the fairness of the strategies for different sets of homogeneous disks, ranging from 8 storage systems up to 8,192 storage systems (see Figure 4). Notice that there are some missing results for Consistent Hashing, Share and Redundant Share. These correspond to configurations that took too much time to evaluate or used more resources than available.

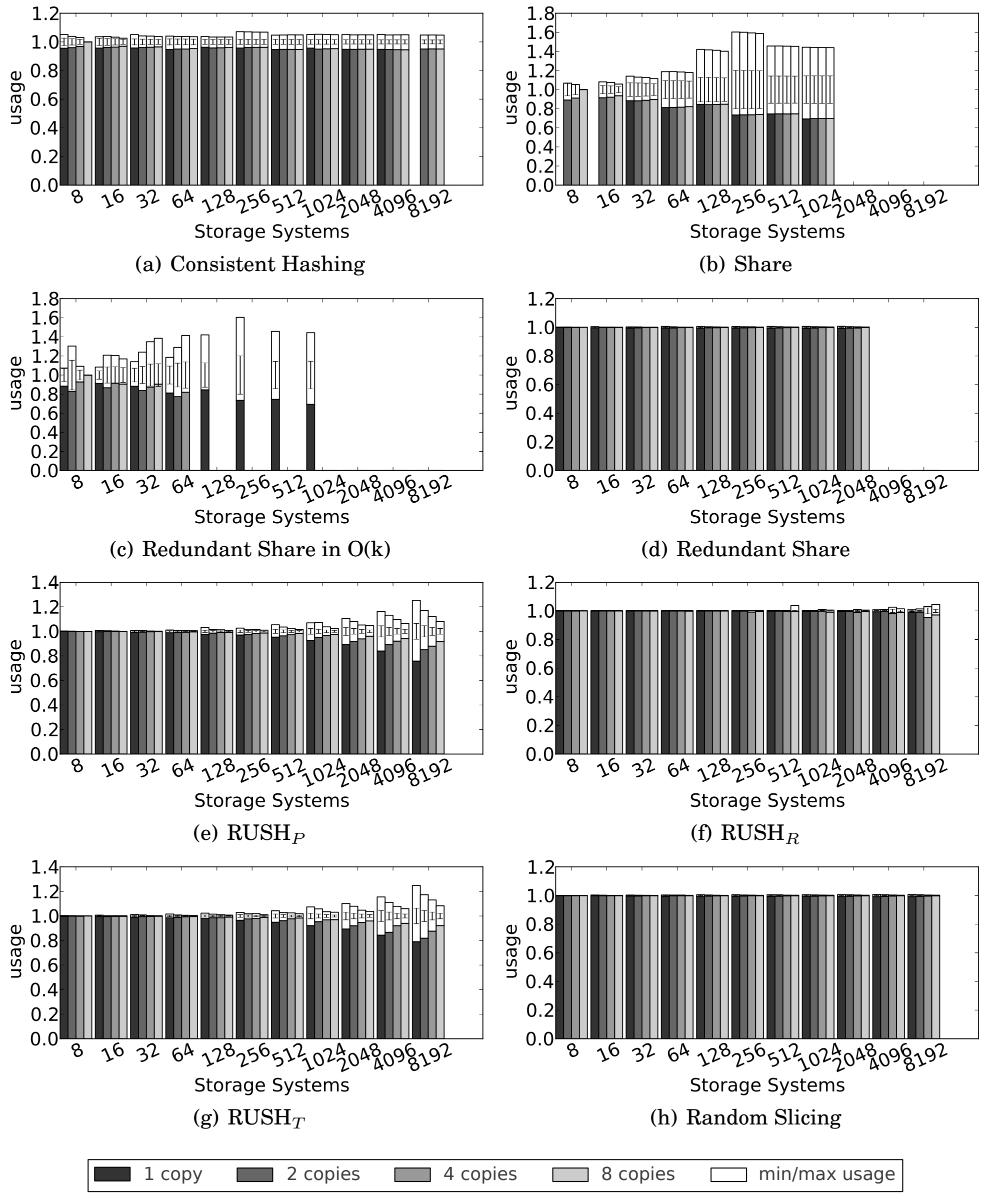
Consistent Hashing has been developed to evenly distribute one copy over a set of homogeneous disks of the same size. Figure 4(a) shows that the strategy is able to fulfill these demands for the test case, in which all disks have the same size. The difference between the maximum and the average usage is always below 7% and the difference between the minimum and average usage is always below 6%. The deviation is nearly independent from the number of copies as well as from the number of disks in the system, so that the strategy can be reasonably well applied. We have thrown $400 \cdot \log n$ points for each storage system (please see Section 2.1 for the meaning of points in Consistent Hashing).

The fairness of Consistent Hashing can be improved by throwing more points for each storage system (see Figure 5 for an evaluation with 64 storage systems). The evaluation shows that initial quality improvements can be achieved with very few additional points, while further small improvements require a high number of extra points per storage system. $400 \cdot \log n$ points are 2400 points for 64 storage systems, meaning that we are already using a high number of points, where further quality improvement becomes very costly.

Share has been developed to overcome the drawbacks of Consistent Hashing for heterogeneous disks. Its main idea is to (randomly) partition the disks into intervals and assign a set of disks to each interval. Inside an interval, each disk is treated as homogeneous and strategies like Consistent Hashing can be applied to finally distribute the data items.

The basic idea implies that Share has to compute and keep the data structures for each interval. 1,000 disks lead to a maximum of 2,000 intervals, implying 2,000 times the memory consumption of the applied uniform strategy. On the other hand, the number of disks inside each interval is smaller than n, which is the number of disks in the environment. The analysis of Share shows that on average $c \cdot \log n$ disks participate in each interval (see Section 2.2, without loss of generality we will neglect the additional $\frac{1}{\delta}$ to keep the argumentation simple). Applying Consistent Hashing as homogeneous strategy therefore leads to a memory consumption, which is in $O(n \cdot \log^2 n \cdot \log^2(\log n))$ and therefore only by a factor of $\log^2(\log n)$ bigger than the memory consumption of Consistent Hashing.

Unfortunately, it is not possible to neglect the constants in a real implementation. Figure 4(b) shows the fairness of Share for a stretch factor of $3 \cdot \log n$, which shows huge deviations even for homogeneous disks. A deeper analysis of the Chernoff-bounds used in [Brinkmann et al. 2002] shows that it would have been necessary to have a



(a) Consistent Hashing
(b) Share
(c) Redundant Share in O(k)
(d) Redundant Share
(e) RUSH$_P$
(f) RUSH$_R$
(g) RUSH$_T$
(h) Random Slicing

Fig. 4. Fairness of the data distribution strategies (homogeneous setting).

stretch factor of 2,146 to keep fairness in the same order as the fairness achieved with Consistent Hashing, which is infeasible in scale-out environments.

Simulations including different stretch factors for 64 storage systems for Share are shown in Figure 6, where the x-axis depicts the stretch factor divided by $\ln n$. The fairness can be significantly improved by increasing the stretch factor. Unfortunately, a stretch factor of 32 already requires in our simulation environment more than 50 GByte main memory for 64 storage systems, making Share impractical in bigger environments. In the following, we will therefore skip this strategy in our evaluations.

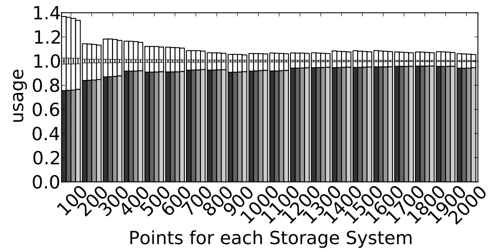


Fig. 5. Influence of point number on Consistent Hashing.

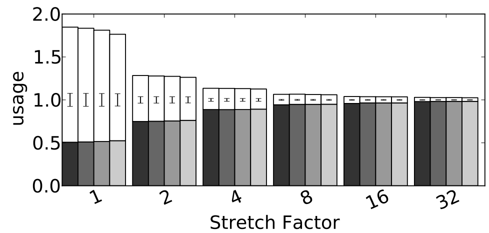


Fig. 6. Fairness of Share depending on stretch factor.

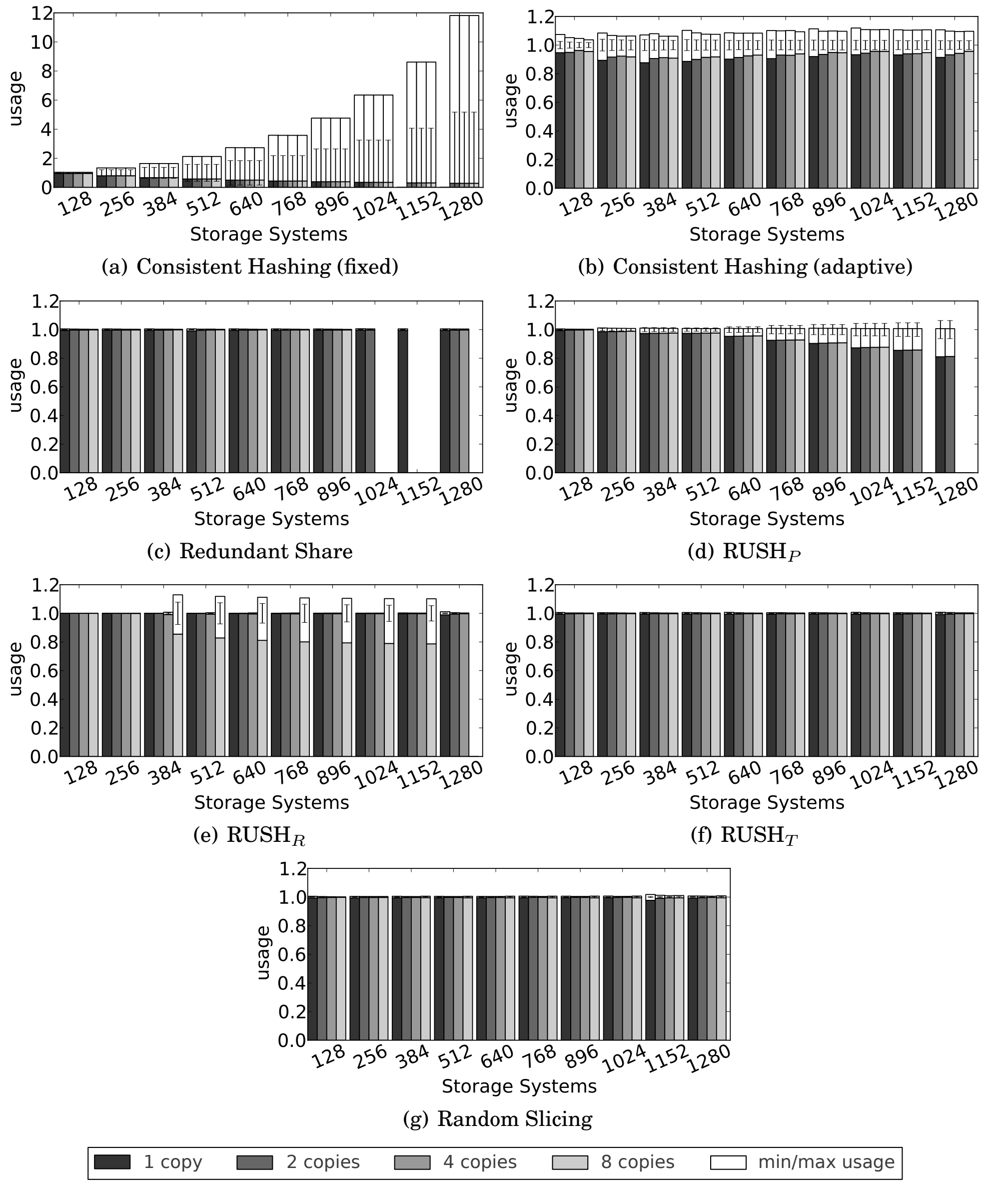
Redundant Share uses precomputed intervals for each disk and therefore does not rely too much on randomization properties. The intervals exactly represent the share of each disk on the total disk capacity, leading to a very even distribution of the data items (see Figure 4(d)). The drawback of this version of Redundant Share is that it has linear runtime, possibly leading to high delays in case of huge environments. Brinkmann et al. have presented enhancements, which enable Redundant Share to have a runtime in $O(k)$, where k is the number of copies [Brinkmann and Effert 2008]. Redundant Share in $O(k)$ requires a huge number of Share instances as sub-routines, making it impractical to support a huge number of disks and a good fairness at the same time. Figure 4(c) shows that it is even difficult to support multiple copies for more than 64 disks, even if the required fairness is low, as 64 GByte main memory have not been sufficient to calculate these distributions. Therefore, we will also neglect Redundant Share with runtime in $O(k)$ in the following measurements.

$RUSH_P$, $RUSH_T$ and $RUSH_R$ place objects almost ideally according to the appropriate weights, though the distribution begins to degrade as the number of disks grows (see Figures 4(e), 4(f) and 4(g)). Interestingly, however, this deviation from the ideal load decreases when the number of copies increases, which might imply that the hash function is not as uniform as expected and needs more samples to provide an even distribution.

In Random Slicing, precomputed partitions are used to represent a disk's share of the total system capacity, in a similar way to Redundant Share's use of intervals. This property, in addition to the hash function used, enforces an almost optimal distribution of the data items, as shown in Figure 4(h).

The fairness of the different strategies for a set of heterogeneous storage systems is depicted in Figure 7. As described in Section 4, we start with 128 storage systems and add every time 128 additional systems having $3/2$-times the capacity of the previously added. Once again, missing results in Redundant Share and $RUSH^*$ are due to configurations too expensive in terms of the computing power available.

The fairness of Consistent Hashing in its original version is obviously very poor (see Figure 7(a)). Assigning the same number of points in the $[0, 1)$-interval for each storage



(a) Consistent Hashing (fixed)

(b) Consistent Hashing (adaptive)

(c) Redundant Share

(d) RUSH$_P$

(e) RUSH$_R$

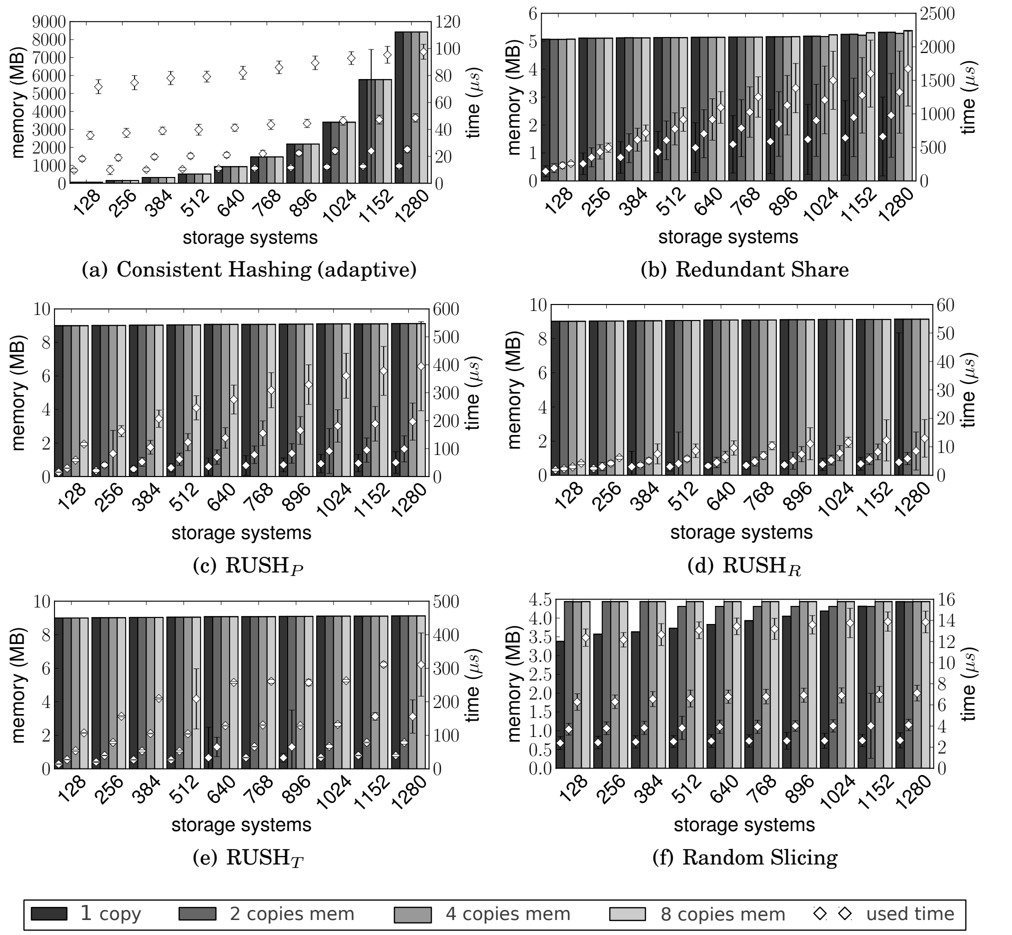
(f) RUSH$_T$

(g) Random Slicing

Fig. 7. Fairness of the data distribution strategies (heterogeneous setting).

system, independent of its size, leads to huge variations. Simply adapting the number of points based on the capacities leads to much better deviations (see Figure 7(b)). The difference between the maximum, respectively minimum and the average usage is around 10% and increases slightly with the number of copies. In the following, we will always use Consistent Hashing with an adaptive number of copies, depending on the capacities of the storage systems.

Both Redundant Share and Random Slicing show again a nearly perfect distribution of data items over the storage systems, due to their precise modeling of disk capacities



(a) Consistent Hashing (adaptive) (b) Redundant Share

(c) RUSH_P (d) RUSH_R

(e) RUSH_T (f) Random Slicing

Fig. 8. Memory consumption and performance of the data distribution strategies (heterogeneous setting).

and the uniformity of the distribution functions (see Figures 7(c) and 7(g), respectively).

The fairness provided by $RUSH_P$ degrades steadily after the fourth reconfiguration. In contrast, $RUSH_T$ delivers a perfect data distribution, even when it was unable to do so in a homogeneous setting (see Figures 7(d) and 7(f), respectively). Figure 7(e), however, shows that $RUSH_R$ does a good distribution job for 1, 2, and 4 copies but degrades with 8 copies showing important deviations from the optimal distribution.

6.2. Memory consumption and compute time

The memory consumption as well as the performance of the different data distribution strategies have a strong impact on the applicability of the different strategies. We assume that scale-out storage systems mostly occur in combination with huge cluster environments, where the different cores of a cluster node can share the necessary data structures for storage management. Assuming memory capacities of 192 GByte per node in 2015 [Amarasinghe et al. 2010], we do not want to waste more than 10% or approximately 20 GByte of this capacity for the metadata information of the underlying storage system. Furthermore, we assume access latencies of 5 ms for magnetic storage systems and access latencies of 50 μs for solid state disks. These access latencies set an upper limit on the time allowed for calculating the translation from a virtual address to a physical storage system.

The bars in Figure 8 represent the average allocated memory, the white bars on top the peak consumption of virtual memory over the different tests. The points in that figure represent the average time required for a single request. These latencies include confidence intervals.

The memory consumption of Consistent Hashing only depends on the number and kind of disks in the system, while the number of copies k has no influence on it (see Figure 8(a)). We are throwing $400 \cdot \log n$ points for the smallest disk, the number of points for bigger disks grows proportional to their capacity, which is necessary to keep fairness in heterogeneous environments. Using 1,280 heterogeneous storage systems requires a memory capacity of nearly 9 GByte, which is still below our limit of 20 GByte.

The time to calculate the location of a data item only depends on the number of copies, as Consistent Hashing is implemented as a $O(1)$-strategy for a single copy. The number of copies has only an influence for a small number of storage systems, e.g., it needs significant more time to place 8 copies if it only uses 8 storage systems. The reason for this is how we chose to implement redundancy: we generate new random values into the $[0, 1)$ ring until we find k different disks, which can take quite long in case of hash collisions. Therefore, this proves that it is possible to use Consistent Hashing in scale-out environments based on solid state drives, as the average latency for the calculation of a single data item stays below 10 μs.

Redundant Share has very good properties concerning memory usage, but the computation time grows linearly in the number of storage systems. Even the calculation of a single item for 128 storage systems takes 145 μs. Using 8 copies increases the average access time for all copies to 258 μs, which is 50 μs for each copy, making it suitable for mid sized environments, that are based on SSDs. Increasing the environment to 1280 storage systems raised the calculation time almost linearly for a single copy to 669 μs, which is reasonable in magnetic disk based environments.

All $RUSH$ variants show good results both in memory consumption and in computation time (see Figures 8(c), 8(d) and 8(e)), being $RUSH_R$ the strategy with the lowest computation time. The reduced memory consumption is explained because the strategies do not need a great deal of in-memory structures in order to maintain the information about clusters and storage nodes. Lookup times depend only on the number of clusters in the system, which can be kept comparatively small for large systems.

Random Slicing shows very good behavior concerning memory consumption and computation time, as both depend only on the number of intervals I currently managed by the algorithm (see Figure 8(f)). In order to compute the position of a data item x, the strategy only needs to locate the interval containing $f_B(x)$, which can be done in $O(\log I)$ using an appropriate tree structure. Furthermore, the algorithm strives to reduce the number of intervals created in each step in order to minimize memory consumption as much as possible. In practice, this yields an average access time of 5 μs for a single data item and 13 μs for 8 copies, while keeping a memory footprint similar to that of Redundant Share.

6.3. Adaptivity

Adaptivity to changing environments is an important requirement for data distribution strategies and one of the main drawbacks of standard RAID approaches. Adding a single disk to a RAID system typically either requires the replacement of all data items in the system or splitting the RAID environment into multiple independent domains.

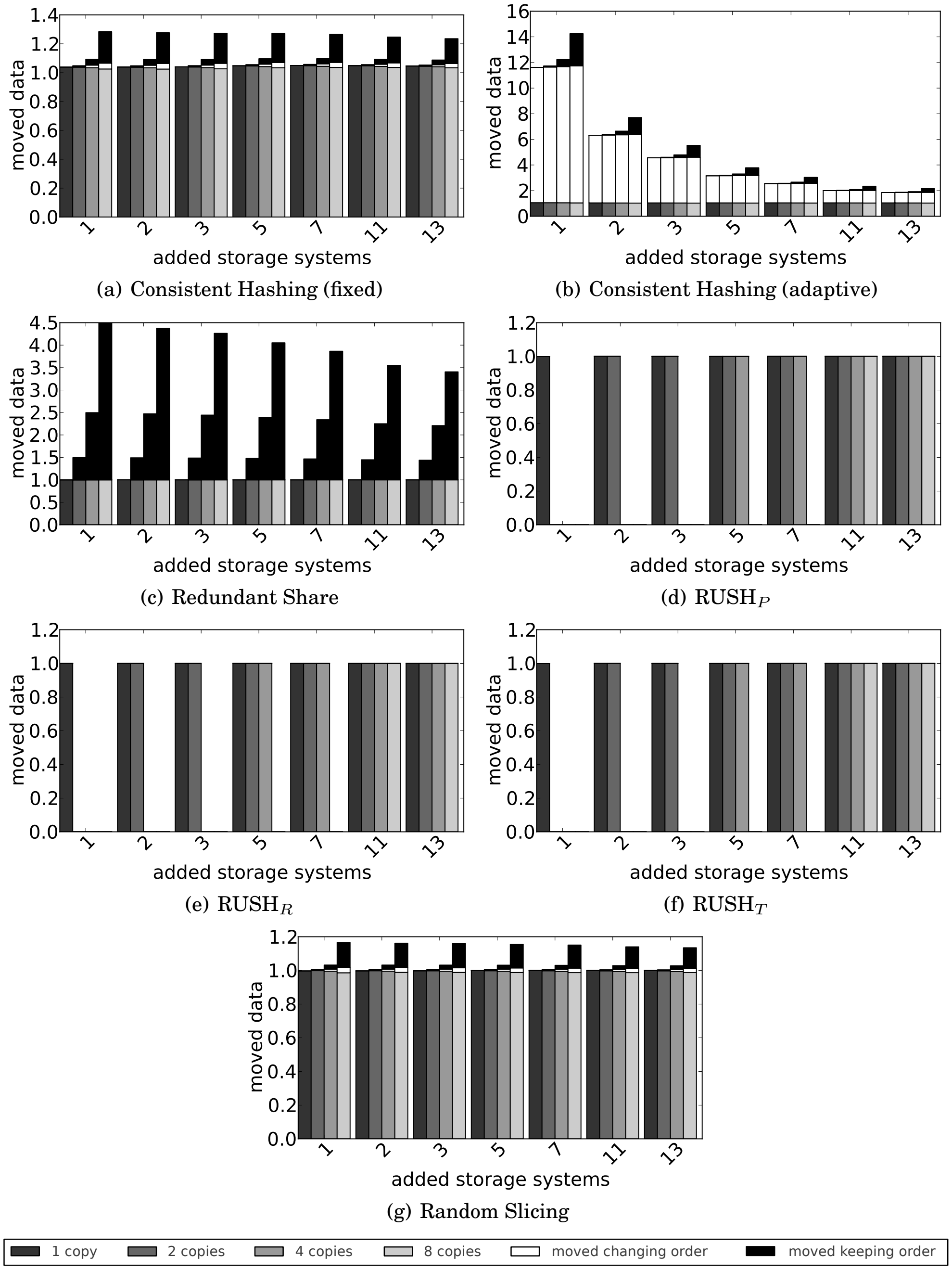


Fig. 9. Adaptivity of the data distribution strategies in a heterogeneous setting.

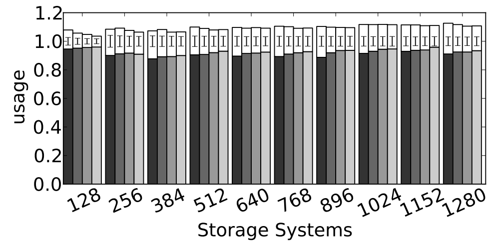


Fig. 10. Fairness of Consistent Hashing for fixed number of points in a heterogeneous setting.

The theory behind randomized data distribution strategies claims that these strategies are able to compete with a best possible strategy in an adaptive setting. This means that the number of data movements to keep the properties of the strategy after a storage system has been inserted or deleted can be bounded against the best possible strategy. We assume in the following that a best possible algorithm just moves as much data from old disks to new disks, respectively from removed disks to remaining disks, as necessary to have the same usage on all storage systems. All bars in Figure 9 have been normalized to this definition of an optimal algorithm.

Furthermore, we distinguish between placements, where the ordering of the data items is relevant and where it is not. The first case occurs, e.g., for standard parity codes, where each data item has a different meaning (labeled "moved keeping order" in Figure 9). If a client accesses the third block of a parity set, then it is necessary to receive exactly that block. In contrast, the second case occurs for RAID 1 sets, where each copy has the same content and receiving any of this blocks is sufficient (labeled "moved changing order"). We will see in the following that not having to keep the order strongly simplifies the rebalancing process.

We start our tests in all cases with 128 storage systems and increase the number of storage systems by 1, 2, 3, 5, 7, 11, or 13 storage systems. The new storage systems have 1,5-times the capacity of the original system.

The original Consistent Hashing paper shows that the number of replacements is optimal for Consistent Hashing by showing that data is only moved from old disks to new disks in case of the insertion of a storage system or from a removed disk to old disks in the homogeneous setting [Karger et al. 1997]. Figure 9(b) shows a very different behavior, the number of data movements is sometimes more than 20-times higher than necessary. The reason is that we are placing $400 \cdot \lceil \log n \rceil$ points for each storage system and $\lceil \log n \rceil$ increases from 7 to 8 when adding storage system number 129. This leads to a large number of data movements between already existing storage systems. Furthermore, the competitiveness strongly depends on whether the ordering of the different copies has to be maintained or not.

Figure 9(a) shows the adaptivity of Consistent Hashing in case that the number of points is fixed for each individual storage system and only depends on its own capacity. We use $2,400$ points for the smallest storage system and use a proportional higher number of points for bigger storage systems. In this case the insertion of new storage systems only leads to data movements from old systems to the new ones and not between old ones and therefore the adaptivity is very good in all cases. Figure 10 shows that the fairness in this case is still acceptable even in a heterogeneous setting.

The adaptivity of Redundant Share for adding new storage systems is nearly optimal, which is in line with the proofs presented in [Brinkmann et al. 2007]. Nevertheless, Redundant Share is only able to achieve an optimal competitiveness if a new storage system is inserted that is at least as big as the previous ones. Otherwise it can happen that Redundant Share is only $\log n$-competitive (see Figure 9(c)).

Table I. Properties of the examined strategies in heterogeneous environments

Strategy	Fairness	Memory Usage	Lookup Time	Adaptivity
Consistent Hashing (fixed)	Poor ($\delta \uparrow$ with n)	N/A	N/A	Good ($\alpha \approx 7\%$)
Consistent Hashing (adapt.)	Moderate ($\delta \approx 10\%$)	High ($\mu \approx$ 8GB)	High ($\tau \approx 98\mu s$)	Poor ($\alpha \approx 1172\%$)
Redundant Share	Good ($\delta \approx 0.36\%$)	Low ($\mu \approx$ 5MB)	Very High ($\tau \approx 1800\mu s$)	Good ($\alpha \approx 0.08\%$)
$RUSH_P$	Poor ($\delta \uparrow$ with n)	Low ($\mu \approx$ 9MB)	Very High ($\tau \approx 400\ \mu s$)	Very Good[1] ($\alpha \approx 0.001\%$)
$RUSH_R$	Good, if $k < 8$ ($\delta \approx 0.22\%$) Poor, if $k = 8$ ($\delta \uparrow$ with n)	Low ($\mu \approx$ 9MB)	Low ($\tau \approx 14\ \mu s$)	Very Good[1] ($\alpha \approx 0.001\%$)
$RUSH_T$	Good ($\delta \approx 0.36\%$)	Low ($\mu \approx$ 9MB)	Very High ($\tau \approx 300\ \mu s$)	Very Good[1] ($\alpha \approx 0.05\%$)
Random Slicing	Good ($\delta \approx 0.4\%$)	Low ($\mu \approx$ 4.5MB)	Low ($\tau \approx 14\ \mu s$)	Good ($\alpha \approx 1.63\%$)

Definitions used: n, number of devices; k, number of copies; δ, average deviation from ideal load; μ, worst case memory consumption; τ, worst case lookup time; α, worst case deviation from ideal number of movements.
[1]The implementation evaluated doesn't support replicas as distinct entities.

Figure 9 shows that $RUSH$ variants adapt nearly optimally when storage nodes are added. Note, however, that we did not evaluate the effect on replica ordering because the current implementations do not support replicas as distinct entities. Instead, $RUSH$ variants distribute all replicas within one cluster. Note also that the missing columns in the results correspond to configurations not accepted in the current implementation.

Figure 9(g) shows that the adaptivity of Random Slicing is very good in all cases. This is explained because intervals for new storage systems are always created from fragments of old intervals, thus forcing data items to migrate only to new storage systems.

6.4. Summary

We conclude this section with a brief qualitative overview of the results collected. Table I summarizes our observations on the evaluated strategies, for all the properties examined. Where available, we provide either average or worst case values of every parameter examined in order to give a general view of each strategy's strong points and weaknesses.

7. INFLUENCE OF THE RANDOMIZATION FUNCTION

As we have seen, one of the most efficient ways to achieve a balanced load is using hashing techniques to distribute data. This relies on the intrinsic connection that exists between randomness and evenness of distribution [Azar et al. 1999] [Raab and Steger 1998]).

In practice, however, real randomness cannot be (easily) achieved, which forces us to rely on pseudo-random number generators (PRNGs) in order to implement the required hashing functions. There are two main requirements which have to be fulfilled by these pseudo-random hash functions:

(1) The computation time for each data value should be as small as possible.

Table II. List of Evaluated PRNGs

PRNG	Type
minstd_rand	LC
minstd_rand0	LC
hellekalek1995	inversive LC
rand48	LC (c != 0)
ecuyer1988	additive combine (LC + LC)
kreutzer1986	shuffle (LC improved)
taus88	XOR LFS
ranlux3	SWC + discard block
ranlux64_3	SWC + discard block
ranlux3_01	SWC + discard block
ranlux64_3_01	SWC + discard block
mt11213b	twisted GFSR
mt19937	twisted GFSR
lagged_fibonacci607	lagged Fibonacci
unix_rand	MC
sha1	SHA-1
md5	MD-5
tiger	Tiger

Algorithm types: Linear congruential (LC), multiplicative congruential (MC), linear feedback shift (LFS), subtract with carry (SWC), general feedback shift register (GFSR).

(2) The pseudo-random hash functions should distribute input data as evenly as possible.

In this section we evaluate how different PRNGs affect the overall performance and fairness of the data distribution strategies examined in this paper.

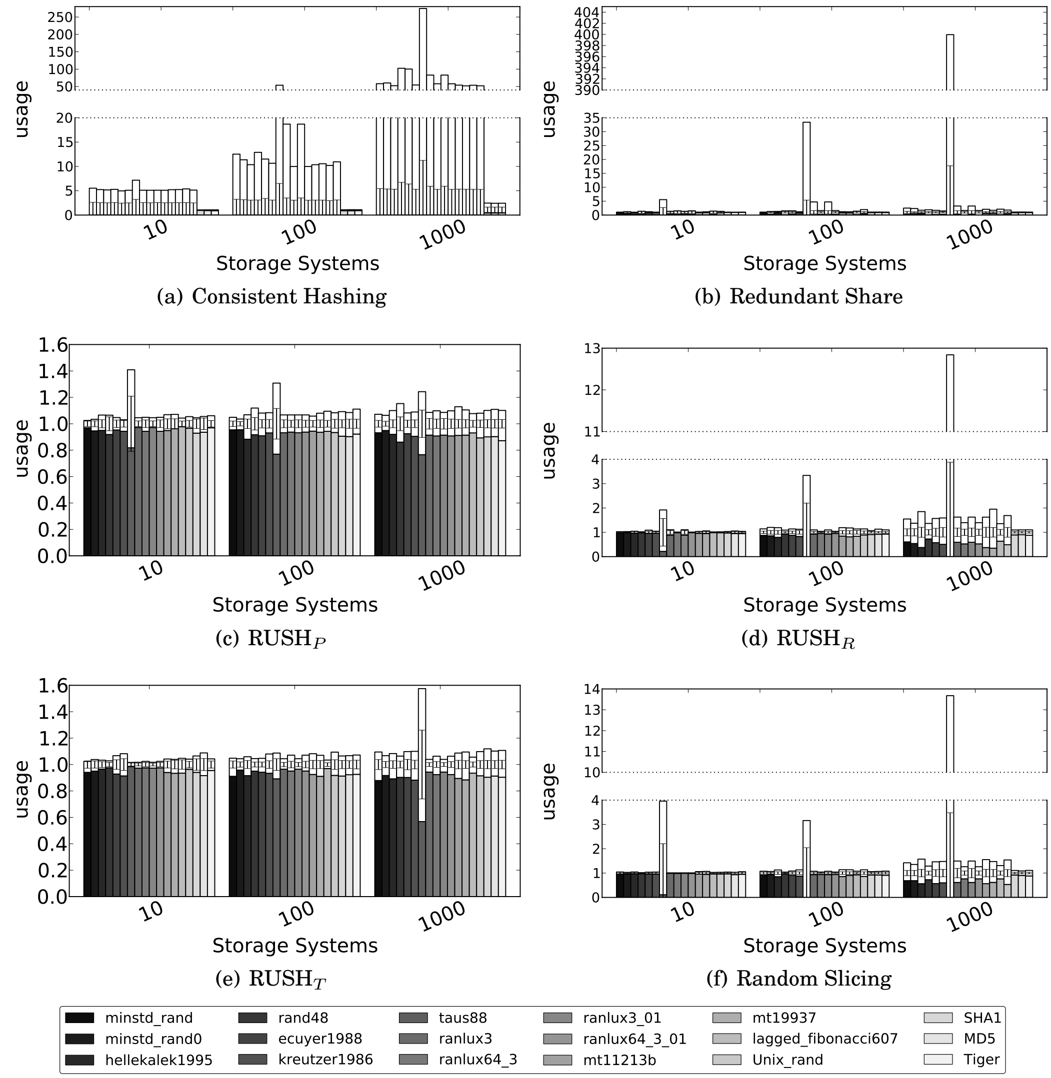
7.1. Experimental Results

Table II lists the set of pseudo-random number generators that we have evaluated. We used implementations from Boost (*www.boost.org*) and Gcrypt (*www.gnu.org/s/libgcrypt*), both well-known and extensively used C++ libraries, and we attempted to cover a diverse set of PRNGs when selecting the algorithms.

These pseudo-random hash functions have been integrated in all of the proposed data distribution strategies. We simulated each strategy with all the aforementioned PRNGs and we measured performance and distribution quality in the same way as the experiments presented in Section 6. In each experiment we distributed $m = 1,000 * n$ balls, with n being the number of configured storage systems. All experiments have been run for 10, 100 and 1,000 homogeneous storage systems.

Figure 11 shows the results obtained when evaluating the fairness provided by each combination of PRNG and strategy. The white boxes in each bar represent the range of results between the minimum and maximum usage per disk. As before, small or non-existing white boxes indicate a very small deviation from an ideal data distribution. Note that for some strategies we needed to split the y-axis to show all the relevant results, due to the huge variability of the measurements obtained with every algorithm.

Interestingly, the results are very different from those presented in Section 6.1. Consistent Hashing, for instance, which showed a distribution that was close to the ideal distribution in the previous experiments, now shows a less-than-ideal data distribution



(a) Consistent Hashing (b) Redundant Share (c) RUSH_P (d) RUSH_R (e) RUSH_T (f) Random Slicing

Fig. 11. Influence of PRNGs on fairness.

with some storage systems even receiving more than 250 times the expected amount of balls in some cases (see Figure 11(a)).

Other strategies seem to be less affected by the change of PRNG, though in general the evaluations show worse results than those obtained with the original hash functions, even for those strategies that previously showed a quasi-ideal distribution like Redundant Share, RUSH_R and Random Slicing (Figures 11(b), 11(d) and 11(f), respectively).

If we consider the generators individually, the results for all the PRNGS evaluated but the last three (*SHA1*, *MD5* and *Tiger*) degrade when increasing the number of storage systems from 100 to 1,000. This effect is less noticeable in Redundant Share (Figure 11(b)), probably because it uses $O(n)$ calls to a PRNG for each ball, which results in a more realistic random distribution.

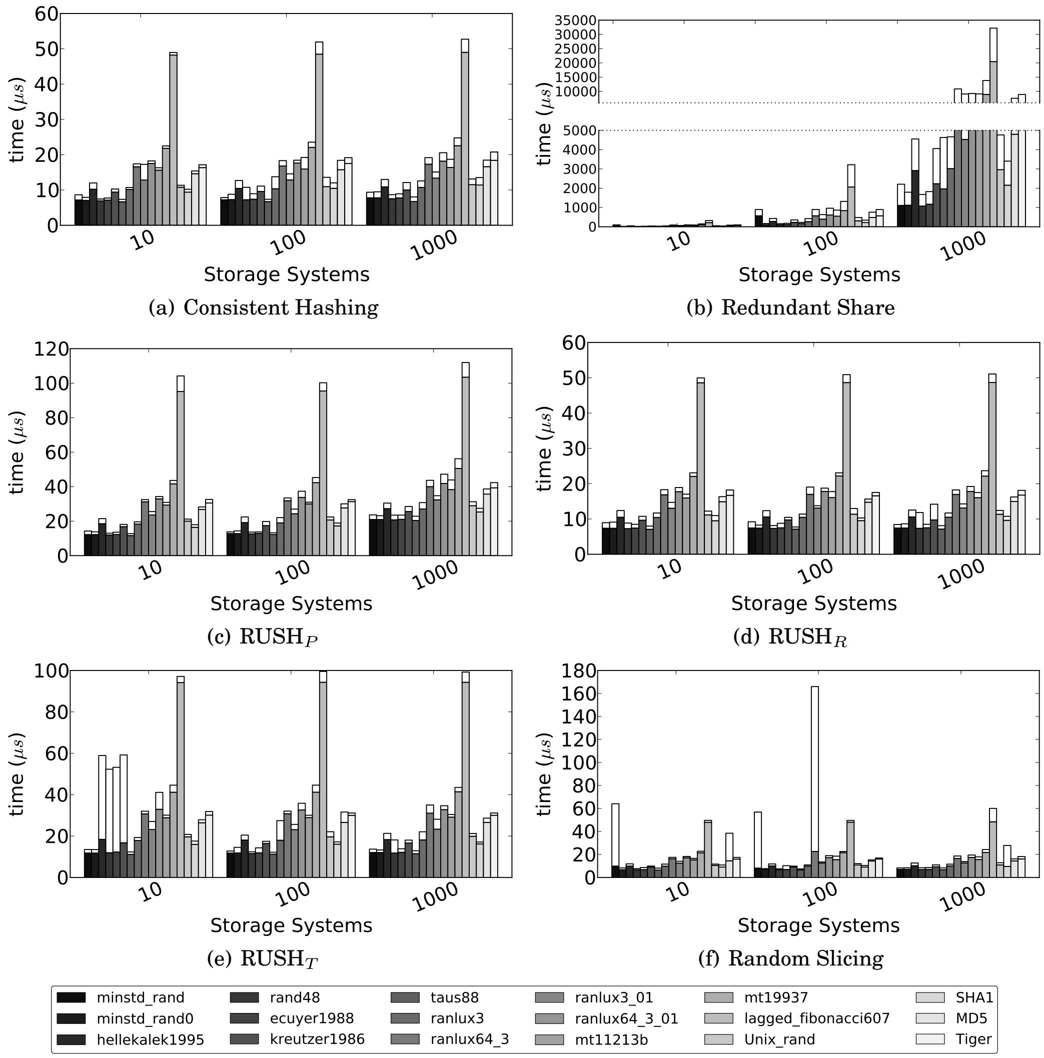


Fig. 12. Influence of PRNGs on performance.

Most interestingly, the *taus88* PRNG shows really bad results in all experiments, significantly degrading the fairness of all strategies when compared against the other PRNGs. This proves that choosing a wrong PRNG can have disastrous results for randomized data distribution strategies, as it can invalidate all the benefits provided by the strategy.

Figure 12 shows the results obtained in the performance experiments for each combination of PRNG and strategy. Each bar shows the average time required for a single request. These times include confidence intervals, which are represented as a white segment in the bar.

Similarly to the results obtained in the previous evaluation, there is a lot of variability in the results obtained. This time, however, the *taus88* PRNG is the one providing the best performance on average for all strategies but Consistent Hashing (see Figure 12(a)), where *rand48* is slightly faster). Notice that in these experiments the *lagged_fibonacci607* generator consistently shows the worst performance results, even though it was able to provide an acceptable fairness.

Notice that all the PRNGs examined tend to behave similarly with 10, 100, or 1,000 storage systems, which is to be expected since the generators are not influenced by the number of devices in the systems. There is one significant exception, however, with the Redundant Share strategy (Figure 12(b)), which shows increased response times in each successive setting. Once again, this can be explained because Redundant Share performs $O(n)$ calls to the selected generator for each data block, which can significantly increase the computation time if n is large.

Most interestingly, notice that PRNGS that provided similar levels of fairness in the previous experiment (e.g., *SHA1*, *MD5*, or *Tiger*), differ significantly when considering average response time. This means that, when designing a randomized distribution strategy, a careful consideration of the selected PRNG is necessary in order to avoid performance or distribution pitfalls.

8. CONCLUSIONS

This paper shows that many randomized data distribution strategies are unable to scale to Exascale environments, as either their memory consumption, their load deviation, or their processing overhead is too high. Nevertheless, they are able to easily adapt to changing environments, a property which cannot be delivered by table- or rule-based approaches.

The proposed Random Slicing strategy combines the advantages of all these approaches by keeping a small table and thereby reducing the amount of necessary random experiments. The presented evaluation and comparison with well-known strategies shows that Random Slicing is able to deliver the best fairness in all the cases studied and to scale up to Exascale data centers.

We have also proven that choosing an appropriate pseudo-random number generator is of the utmost importance when designing and evaluating a new randomized data distribution strategy, since there is a significant risk of degrading the quality of the strategy.

REFERENCES

AMARASINGHE, S., CAMPBELL, D., CARLSON, W., CHIEN, A., DALLY, W., ELNOHAZY, E., HALL, M., ET AL. 2010. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Tech. rep., sponsored by DARPA IPTO in the context of the ExaScale Computing Study.

AZAGURY, A., DREIZIN, V., FACTOR, M., HENIS, E., NAOR, D., RINETZKY, Y., RODEH, O., SATRAN, J., TAVORY, A., AND YERUSHALMI, L. 2003. Towards an object store. In *Proceedings of the 20^{th} IEEE Conference on Mass Storage Systems and Technologies (MSST)*. 165–176.

AZAR, Y., BRODER, A., KARLIN, A., AND UPFAL, E. 1999. Balanced allocations. *SIAM journal on computing 29,* 1, 180–200.

BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. 1994. EVENODD: an optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21^{st} International Symposium on Computer Architecture (ISCA)*. 245–254.

BRINKMANN, A. AND EFFERT, S. 2008. Redundant data placement strategies for cluster storage environments. In *Proceedings of the 12^{th} International Conference On Principles Of DIstributed Systems (OPODIS)*.

BRINKMANN, A., EFFERT, S., MEYER AUF DER HEIDE, F., AND SCHEIDELER, C. 2007. Dynamic and Redundant Data Placement. In *Proceedings of the 27^{th} IEEE International Conference on Distributed Computing Systems (ICDCS)*. Toronto, Canada.

BRINKMANN, A., HEIDEBUER, M., MEYER AUF DER HEIDE, F., RÜCKERT, U., SALZWEDEL, K., AND VODISEK, M. 2004. V: Drive - costs and benefits of an out-of-band storage virtualization system. In *Proceedings of the 21^{st} IEEE Conference on Mass Storage Systems and Technologies (MSST)*. 153–157.

BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER, C. 2000. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the* 12^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 119–128.

BRINKMANN, A., SALZWEDEL, K., AND SCHEIDELER, C. 2002. Compact, adaptive placement schemes for non-uniform distribution requirements. In *Proceedings of the* 14^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. Winnipeg, Manitoba, Canada, 53–62.

CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. 2004. Row-diagonal parity for double disk failure correction. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST)*. San Francisco, CA, 1–14.

CORTES, T. AND LABARTA, J. 2001. Extending heterogeneity to RAID level 5. In *Proceedings of the USENIX Annual Technical Conference*. Boston, Massachusetts, 119–132.

DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review 41,* 6, 205–220.

DEVULAPALLI, A., DALESSANDRO, D., AND WYCKOFF, P. 2008. Data Structure Consistency Using Atomic Operations in Storage Devices. In *Proceedings of the* 5^{th} *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. Baltimore, USA, 65 – 73.

EASTLAKE, D. AND JONES, P. 2001. US secure hash algorithm 1 (SHA1).

GONZALEZ, J. AND CORTES, T. 2008. Distributing Orthogonal Redundancy on Adaptive Disk Arrays. In *Proceedings of the International Conference on Grid computing, high-performAnce and Distributed Applications (GADA)*. Monterrey, Mexico.

HONICKY, R. J. AND MILLER, E. L. 2003. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the* 17^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Nice, France.

HONICKY, R. J. AND MILLER, E. L. 2004. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In *Proceedings of the* 18^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

JOHNSON, N. L. AND KOTZ, S. 1977. *Urn Models and Their Applications*. John Wiley and Sons, New York.

KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the* 29^{th} *ACM Symposium on Theory of Computing (STOC)*. El Paso, Texas, USA, 654–663.

LUBY, M. 1996. *Pseudorandomness and Cryptographic Applications*. Princeton University Press.

MENSE, M. AND SCHEIDELER, C. 2008. Spread: An adaptive scheme for redundant and fair storage in dynamic heterogeneous storage systems. In *Proceedings of the* 19^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. San Francisco, California.

MIRANDA, A., EFFERT, S., KANG, Y., MILLER, E., BRINKMANN, A., AND CORTES, T. 2011. Reliable and randomized data distribution strategies for large scale storage systems. In *High Performance Computing (HiPC), 2011 18th International Conference on*. IEEE, 1–10.

MITZENMACHER, M. 1996. The power of two choices in randomized load balancing. Ph.D. thesis, Computer Science Department, University of California at Berkeley.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*. 109–116.

POPOV, I., BRINKMANN, A., AND FRIEDETZKY, T. 2012. On the Influence of PRNGs on Data Distribution. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*. IEEE, 536–543.

RAAB, M. AND STEGER, A. 1998. balls into binsa simple and tight analysis. *Randomization and Approximation Techniques in Computer Science*, 159–170.

SANDERS, P. 2001. Reconciling simplicity and realism in parallel disk models. In *Proceedings of the* 12^{th} *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, Philadelphia, PA, 67–76.

SCHINDELHAUER, C. AND SCHOMAKER, G. 2005. Weighted distributed hash tables. In *Proceedings of the* 17^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. Las Vegas, Nevada, USA, 218–227.

STEVENS, M., LENSTRA, A., AND DE WEGER, B. 2007. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In *Proceedings of the* 26^{th} *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*. 1–22.

STEVENS, M., SOTIROV, A., APPELBAUM, J., LENSTRA, A., MOLNAR, D., OSVIK, D., AND DE WEGER, B. 2009. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Proceedings of the* 29^{th} *Annual International Cryptology Conference (CRYPTO)*. 55–69.

STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M., DABEK, F., AND BALAKRISHNAN, H. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking 11,* 1, 17–32.

VIEGA, J. 2003. Practical random number generation in software. In *Proceedings of the* 19^{th} *Annual Computer Security Applications Conference (ACSAC)*. 129–141.

WANG, T. 2007. Integer hash function. *Online: http://www. concentric. net/ ttwang/tech/inthash. htm*.

WANG, X., YIN, Y., AND YU, H. 2005. Finding collisions in the full sha-1. In *Proceedings of the* 25^{th} *Annual International Cryptology Conference (CRYPTO)*. 17–36.

WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. 2006a. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, USA, 307–320.

WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. 2006b. CRUSH: Controlled, Scalable And Decentralized Placement Of Replicated Data. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. Tampa, FL.

ZHENG, W. AND ZHANG, G. 2011. Fastscale: Accelerate raid scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*.

Received Month Year; revised Month Year; accepted Month Year